# Теоретические вопросы по лабораторной работе №3

1. [Шаблон проектирования MVC.](#шаблоныПроектированияMVC)
2. [Луковая архитектура.](#луковаяАрхитектура)
3. [Технология gRPC. Отличие REST, gRPC](#gRPC_отличияREST_gRPC)
4. [Технология GraphQL. Отличие REST, GraphQL](#GraphQL_отличияREST_GraphQL)
5. [Обзор серверных фреймворков: JavaScript, Python, C#](#обзорСерверныхФреймворков_js_python_cs)
6. [Обзор серверных фреймворков: Java, PHP, Ruby](#обзорСерверныхФреймворков_java__ruby)
7. [Doker. Цель технологии. Основные сведения. Термины и концепции. Файл Dockerfile.](#dockerЦельТехнологииОснСведения_прочаяХу)
8. [Doker. Цель технологии. Основные сведения. Команды. Работа с данными. Kubernetes. Сравнение Kubernetes и Docker Swarm.](#хуита8)

# [Шаблон проектирования MVC](#_top)

## MVC

MVC расшифровывается как «модель-представление-контроллер» (от англ. model-view-controller). Это способ организации кода, который предполагает выделение блоков, отвечающих за решение разных задач (для более простой работы с ним). Один блок отвечает за данные приложения, другой отвечает за внешний вид, а третий контролирует работу приложения.

## Контроллер

В модели MVC самым главным элементом, с которого все начинается и на котором все, как правило, заканчивается - это [контроллер](https://webkyrs.info/page/symfony-controller-chto-eto-i-kak-ego-sozdat).

Задача контроллера - принять запрос пользователя и решить, что делать с этим запросом далее, если требуется, то перенаправить запрос в модель, который обработает информацию и возвращает ее контроллеру.

После того, как информация обработана, контроллер решает, что с ней делать дальше, если пользователю достаточно предоставить просто набор каких-то данных, не в виде html-страницы, а например, в формате json, контроллер этот набор данных ему выдает.

Если необходимо сформировать html-страницу, контроллер передает эти данные в вид и внутри вида шаблонизатор формирует каркас страницы, выдает ее назад контроллеру и контроллер уже выдает этот каркас пользователю в виде html-страницы.

Из 3 частей MVC модели контроллер является обязательной частью. Остальные части являются опциональными. Если пользователю достаточно только отдать какой-то набор данных, то можно обойтись без вида. Если не нужно обрабатывать данные, то можно обойтись без модели.

Контроллер - это некоторая функция. Может содержать ссылки на какие-то другие функции и возвращает какой-то результат.

Контроллер можно сравнить с начальником отдела на предприятии. Начальнику отдела приходит задание от директора предоставить какой-то отчет и далее контроллер, под которым мы имеем в виду начальника отдела, принимает решение. Или он сам сделает этот отчет и отдаст его директору или он может передать ее выполнение своим подчиненным. В качестве подчиненного можно представить "модель".

## Модель

[Модель](https://webkyrs.info/page/chto-takoe-servisy-v-symfony)- это "подчиненный", который делает какую-то рутинную работу, выполняет обработку данных. Он произвел эту обработку данных и выдал результат начальнику (контроллеру). Контроллер принял этот отчет.

После того, как данные обработаны, начальник (контроллер) относит этот отчет директору (т.е. человеку, который запросил эти данные).

Примерно на таком же принципе работает и модель MVC.

**Модель** - это данные и правила, которые используются для работы с данными, которые представляют концепцию управления приложением. В любом приложении вся структура моделируется как данные, которые обрабатываются определённым образом.

Модель даёт контроллеру представление данных, которые запросил пользователь (сообщение, страницу книги, фотоальбом, и тому подобное). Модель данных будет одинаковой, вне зависимости от того, как мы хотим представлять их пользователю. Поэтому мы выбираем любой доступный вид для отображения данных.

Модель содержит наиболее важную часть логики нашего приложения, логики, которая решает задачу, с которой мы имеем дело (форум, магазин, банк, и тому подобное). Контроллер содержит в основном организационную логику для самого приложения (очень похоже на ведение домашнего хозяйства).

## Вид

[Вид](https://webkyrs.info/page/chto-takoe-shablonizator-twig-i-zachem-on-nuzhen)- можно представить тоже в виде каких-то работников отдела, которые набирают уже подготовленные данные от модели на компьютере, например в текстовом редакторе Microsoft Word и распечатывают его. Вот, по сути,  этом и заключается задача "вида".

Задача модели - это такие работники, которые производят расчеты, расчетчики какие-то и.т.д. Примерно такую аналогию можно произвести.

Модель - это работник или исполнитель. Содержит основные функции и классы (логику).

**Вид** обеспечивает различные способы представления данных, которые получены из модели. Он может быть шаблоном, который заполняется данными. Может быть несколько различных видов, и контроллер выбирает, какой подходит наилучшим образом для текущей ситуации.

Веб приложение обычно состоит из набора контроллеров, моделей и видов. Контроллер может быть устроен как основной, который получает все запросы и вызывает другие контроллеры для выполнения действий в зависимости от ситуации.

## Преимущества

* это чёткое разделение логики представления (интерфейса пользователя) и логики приложения. Поддержка различных типов пользователей, которые используют различные типы устройств является общей проблемой наших дней. Предоставляемый интерфейс должен различаться, если запрос приходит с персонального компьютера или с мобильного телефона. Модель возвращает одинаковые данные, единственное различие заключается в том, что контроллер выбирает различные виды для вывода данных.
* Существенно уменьшает сложность больших приложений. Код получается гораздо более структурированным, и, тем самым, облегчается поддержка, тестирование и повторное использование решений.

## А зачем использовать рабочую среду?

Когда вы используете рабочую среду, базовая структура MVC уже подготовлена, и вам остаётся только расширить структуру, размещая ваши файлы в соответствующих директориях для соответствия шаблону MVC. Кроме того, у вас будет набор функций, которые уже написаны и хорошо протестированы.

Другие шаблоны проектирования:

1. MVP (Model-View-Presenter) - архитектура, в которой присутствуют три основных компонента: модель, представление и презентер. Презентер обрабатывает пользовательский ввод и управляет изменением модели и представления.

2. MVVM (Model-View-ViewModel) - архитектура, которая разделяет приложение на три компонента: модель, представление и ViewModel. ViewModel является посредником между моделью и представлением, управляя отображением данных и обработкой пользовательского ввода.

3. Clean Architecture - архитектура, которая разделяет приложение на слои, каждый из которых имеет свою собственную ответственность. Она позволяет легко добавлять новые функции и изменять существующие без влияния на другие части приложения.

4. Flux - архитектура, которая используется в React-приложениях для управления состоянием. Она разделяет приложение на четыре компонента: диспетчер, хранилище, представление и действия. Действия вызываются пользователем и передаются диспетчеру, который обновляет хранилище и уведомляет представление об изменении состояния.

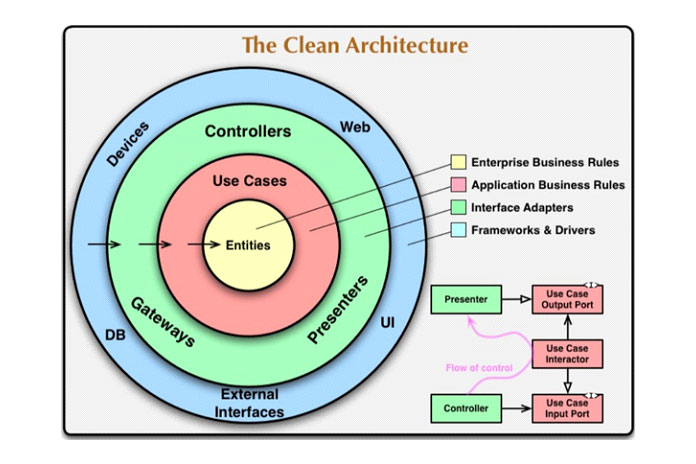
5. Domain-driven design (DDD) - архитектурный подход, который сосредоточен на бизнес-логике приложения. Он разделяет приложение на слои, каждый из которых имеет свою собственную ответственность. Он также подчеркивает важность языка бизнеса и учитывает контекст, в котором используется приложение.

[Ссылка в начало](#_top)

# [Луковая архитектура](#_top)

**Onion**- архитектура представляет собой разделение приложения на уровни. При чем есть один независимый уровень, который находится в центре архитектуры. От этого уровня зависит второй уровень, от второго - третий и т.д. То есть получается, что вокруг первого независимого уровня наслаивается второй-зависимый. Вокруг второго наслаивается третий, который также может зависеть и от первого. Образно это может быть выражено в виде лука, в котором также есть сердцевина, вокруг которого наслаиваются все остальные слои, вплоть до шелухи.

Количество уровней может отличаться, но в центре всегда находится модель домена (Domain Model), то есть те классы моделей, которые используются в приложении и объекты которых хранятся в базе данных:



Первый уровень вокруг модели домена образуют интерфейсы, которые управляют работой с моделью домена. Обычно это интерфейсы репозиториев, через которые мы взаимодействуем с базой данных.

Внешний уровень представляет такие компоненты, которые очень часто изменяются. Обычно внешний уровень образуют пользовательский интерфейс, тесты, какие-то вспомогательные классы инфраструктуры приложения. К этому уровню также относятся конкретные реализации интерфейсов, объявленных на нижележащих уровнях. Например, реализация интерфейса репозитория, который объявлен на уровне Domain Services. Вообще все внутрение уровни, которые можно объединить в Application Core, определяют только интерфейсы, а конкретная реализация этих интерфейсов располагается на внешнем уровне.

Также стоит отметить, что все внешние хранилища, как базы данных, файлы, внешние веб-сервисы, от которых мы можем получать данные, - все это является внешним по отношению к архитектуре.

[Ссылка в начало](#_top)

# [Технология gRPC. Отличие REST, gRPC](#_top)

## Что такое gRPC?

gRPC – это архитектура и система API с открытым исходным кодом, управляемая Cloud Native Computing Foundation. Она основана на модели удаленного вызова процедур (RPC). Хотя модель RPC обширна, gRPC обладает особой реализацией.

## ****Что такое RPC?****

В RPC взаимодействие между клиентом и сервером происходит так, как если бы клиентский API-запрос был локальной операцией или запрос был внутренним кодом сервера.

В RPC клиент отправляет запрос процессу на сервере, который постоянно прослушивает удаленные вызовы. В запросе содержится вызываемая серверная функция и все передаваемые параметры. RPC API использует в качестве базового механизма обмена данными протокол типа HTTP, TCP или UDP.

## ****Чем gRPC отличается от RPC?****

gRPC – это система, реализующая традиционный RPC с несколькими оптимизациями. Например, gRPC использует Protocol Buffers и HTTP 2 для передачи данных.

Она также абстрагирует механизм обмена данными от разработчика. Например, другая широко распространенная реализация RPC API, OpenAPI, требует от разработчиков сопоставления концепций RPC с протоколом HTTP. Но gRPC абстрагирует базовую HTTP-связь. Эти оптимизации делают gRPC быстрее, проще в реализации и удобнее для работы в Интернете по сравнению с другими реализациями RPC.

## Что такое REST?

REST – это подход к архитектуре программного обеспечения, определяющий набор правил для обмена данными между программными компонентами. Он основан на HTTP, стандартном протоколе связи в Интернете. Интерфейсы RESTful API управляют связью между клиентом и сервером с помощью HTTP-команд, таких как POST, GET, PUT и DELETE, для операций создания, чтения, обновления и удаления. Ресурс на стороне сервера идентифицируется по адресу в виде URL.

REST работает следующим образом:

1. Клиент делает запрос на создание, изменение или удаление ресурса на сервере.
2. Запрос содержит адрес ресурса, а также может включать дополнительные параметры.
3. Сервер отвечает, возвращая клиенту весь ресурс после завершения операции.
4. Ответ содержит данные в формате JSON и коды состояния.

Интерфейсы API, созданные с использованием рекомендаций REST, называются RESTful API или REST API.

## Почему организации используют gRPC и REST?

gRPC и REST – это два разных подхода к разработке API.

Работа с API аналогична заказу блюд в ресторане через меню. В любом ресторане клиент (заказчик) может заказать еду из меню (API), которое имеет фиксированный набор блюд. Эта информация передается на кухню (сервер), которая готовит запрошенное блюдо и выдает его клиенту. Клиенту нужно знать, не как кухня выполняет заказ, а только конечный результат. Стандартизация форматов меню означает, что клиенты и кухни знают, как их использовать.

Без API не было бы общего соглашения о том, как взаимодействуют различные приложения или программные сервисы. Программисты двух разных приложений должны были бы каждый раз общаться друг с другом, чтобы определить способ обмена данными.

Существуют различные типы архитектур API, такие как gRPC и REST, поскольку те или иные могут лучше подходить для различных примеров использования в организации. Разработчик API должен выбрать предпочтительную клиент-серверную архитектуру, исходя из системных требований.

## В чем сходство gRPC и REST?

REST и gRPC имеют некоторые общие черты архитектурных подходов к API.

### ****Механизм обмена данными****

Обе системы позволяют двум программным компонентам, клиенту и серверу, взаимодействовать и обмениваться данными на основе общего набора правил. Эти правила применяются независимо от внутренней работы каждого программного компонента.

**Связь на основе HTTP**

Обе системы передают данные через механизм запроса-ответа HTTP, который является предпочтительным эффективным протоколом связи в Интернете. Однако в gRPC это скрыто от разработчика, а в REST это более очевидно.

### ****Гибкость реализации****

Как REST, так и gRPC можно реализовать на широком спектре языков программирования. Благодаря этому они хорошо переносятся из одной среды программирования в другую, что обеспечивает оптимальную совместимость с почти универсальной поддержкой.

### ****Пригодность для масштабируемых распределенных систем****

И gRPC, и REST используют следующее:

* Асинхронную связь, благодаря которой клиент и сервер могут взаимодействовать, не прерывая операции.
* Архитектуру без фиксации состояния, поэтому серверу не нужно запоминать состояние клиента.

Это означает, что разработчики могут использовать gRPC и REST для создания отказоустойчивых систем с большим количеством одновременных запросов. Можно создавать масштабируемые распределенные системы с несколькими клиентами.

### Принципы архитектуры gRPC и REST

Хотя REST и gRPC имеют схожие функции, базовые модели значительно отличаются по своей архитектуре.

### ****Коммуникационная модель****

При использовании REST API клиент отправляет на сервер один запрос REST API, а сервер на это отправляет один ответ. Клиент должен дождаться ответа от сервера, прежде чем продолжить работу. Этот механизм представляет собой модель «запрос-ответ» и является одиночной передачей данных (один-к-одному).

При использовании же gRPC клиент может отправить один или несколько API-запросов на сервер, которые могут привести к одному либо нескольким ответам от сервера. Передача данных может быть односторонней (один-к-одному), потоковой (один-ко-многим), потоковой от клиента (многие-к-одному) или двунаправленной (многие-ко-многим). Этот механизм представляет собой коммуникационную модель «ответ для передачи на сторону клиента» и возможен благодаря тому, что gRPC работает на основе HTTP 2.

### ****Вызов операций на сервере****

В gRPC API вызов операций на сервере осуществляется сервисами, также известными как функции или процедуры. Клиент gRPC вызывает эти функции так же, как и вы внутри приложения. Это называется сервис-ориентированной архитектурой. Вот пример.

createNewOrder(customer\_id, item\_id, item\_quantity) -> order\_id.

В REST существует ограниченный набор HTTP-команд, которые клиент может использовать на ресурсах сервера, определенных URL-адресом. Клиент сам вызывает ресурс. Это называется субъект-ориентированной архитектурой. Субъект-ориентированная архитектура хорошо сочетается с методами объект-ориентированного программирования. Вот пример.

POST /orders <headers> (customer\_id, item\_id, item\_quantity) -> order\_id.

Хотя gRPC API можно разрабатывать на основе субъект-ориентированного подхода, это не является ограничением самой системы.

### ****Формат обмена данными****

При использовании REST API структуры данных, передаваемые между программными компонентами, обычно выражаются в формате обмена данными JSON. Можно передавать другие форматы данных, такие как XML и HTML. JSON легко читается и является гибким, хотя его необходимо сериализовать и переводить на язык программирования.

В отличие от этого, gRPC по умолчанию использует формат Protocol Buffers (Protobuf), хотя в нем также имеется встроенная поддержка JSON. Сервер определяет структуру данных, используя язык описания интерфейса Protocol Buffer (IDL) в файле протоспецификации. Затем gRPC сериализует структуру в двоичный формат и десериализует ее на любой заданный язык программирования. Этот механизм работает быстрее, чем использование JSON, который не сжимается во время передачи. Формат Protocol Buffers не читабельный для человека, в отличие от JSON, используемого REST API.

### Другие ключевые отличия gRPC и REST

Помимо архитектурного стиля, gRPC и REST имеют и другие присущие отличия.

### ****Взаимозависимость «клиент-сервер»****

REST имеет слабую взаимосвязь, то есть клиент и сервер не должны ничего знать о реализации друг друга. Такая взаимозависимость облегчает доработку API со временем, поскольку изменение определений сервера не обязательно требует изменения кода на стороне клиента.

gRPC имеет сильную взаимосвязь, то есть клиент и сервер должны иметь доступ к одному и тому же файлу PROTO. Любые обновления файла требуют обновлений как на стороне сервера, так и на стороне клиента.

### ****Генерация кода****

gRPC имеет встроенный набор функций генерации внутреннего кода на сторонах клиента и сервера. Они доступны на нескольких языках благодаря протоколу-компилятору Protocol Buffers. После определения структуры в файле PROTO система gRPC генерирует код на сторонах клиента и сервера. Генерация кода делает разработку API менее затратной по времени.

Однако REST не предлагает встроенных механизмов генерации кода, поэтому разработчики должны использовать дополнительные инструменты сторонних производителей, если им нужна эта функция.

### ****Двунаправленная трансляция данных****

gRPC обеспечивает двунаправленную потоковую передачу данных. Это означает, что и клиент, и сервер могут одновременно посылать и получать несколько запросов и ответов в рамках одного соединения.

REST не имеет этой функции.

## Когда использовать gRPC, а когда – REST

В настоящее время REST является самой популярной архитектурой API для веб-сервисов и микросервисных архитектур. Популярность REST обусловлена простотой реализации и отображения структуры данных, а также удобством чтения и гибкостью. Начинающим программистам несложно начать разрабатывать интерфейсы RESTful API для своих приложений, будь то для веб-сервисов или внутренних микросервисов.

Примеры использования REST API:

* Веб-архитектуры.
* Общедоступные интерфейсы API для легкого понимания внешними пользователями.
* Простой обмен данными.

Система gRPC, в отличие от REST, была разработана специально для того, чтобы дать разработчикам возможность создавать высокопроизводительные API для микросервисных архитектур в распределенных центрах обработки данных. Она лучше подходит для внутренних систем, требующих потоковой передачи данных в реальном времени и загрузки больших объемов информации. gRPC также хорошо подходит для микросервисных архитектур на нескольких языках программирования, для которых API вряд ли будет меняться со временем.

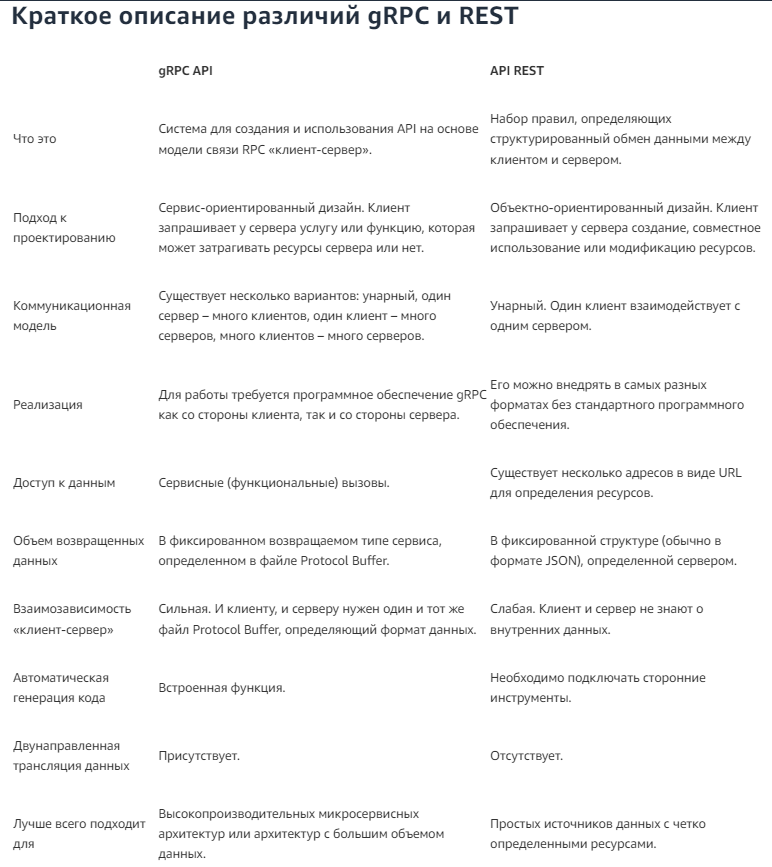
gRPC API лучше использовать в следующих случаях:

* Создание высокопроизводительных систем.
* Загрузка больших данных.
* Разработка приложений реального времени или потоковых приложений.

### ****Заметка о разработке сетевого программного обеспечения****

Хотя HTTP является основным веб-протоколом, существуют разные версии HTTP, которые в разной степени используются в веб-браузерах и веб-серверах.

gRPC API всегда использует версию HTTP 2, а REST API обычно – версию HTTP 1.1, которая отличается от протокола HTTP. Несмотря на то, что версия HTTP 2 стала общепринятым веб-протоколом, она не имеет универсальной поддержки браузеров, в отличие от HTTP 1.1. Такая ограниченная поддержка может сделать gRPC менее привлекательным вариантом для разработчиков, желающих обеспечивать работу веб-приложений.



[Ссылка в начало](#_top)

# [Технология GraphQL. Отличие REST, GraphQL](#_top)

## В чем разница между GraphQL и REST?

GraphQL и REST – два разных подхода к разработке API для обмена данными через Интернет. REST дает возможность клиентским приложениям обмениваться данными с сервером с помощью команд HTTP – стандартного протокола связи в Интернете. А GraphQL – это язык запросов API, определяющий спецификации, по которым клиентское приложение должно запрашивать данные с удаленного сервера. Вы можете использовать GraphQL в вызовах API для определения запроса, не полагаясь на серверное приложение. GraphQL и REST – две мощные технологии, на которых основывается большинство наших современных приложений

## В чем сходство между GraphQL и REST?

И GraphQL, и REST – популярные стили архитектуры API, которые позволяют обмениваться данными между различными сервисами или приложениями в модели «клиент-сервер».

С помощью API можно получить доступ к данным и выполнить такие операции с ними:

1. Клиент отправляет запрос API одному или нескольким адресам на сервере
2. Сервер выдает ответ, содержащий данные, состояние данных или коды ошибок

REST и GraphQL позволяют создавать, изменять, обновлять и удалять данные в отдельном приложении, сервисе или модуле через API. API, разработанные с помощью REST, известны как API RESTful или API REST. Те, которые разработаны с помощью GraphQL, представляют собой обычные API GraphQL.

Команды разработчиков внешнего и внутреннего интерфейса используют эти архитектуры API для создания модульных и доступных приложений. Использование архитектуры API помогает обеспечить безопасность, модульность и масштабируемость систем, а также делает системы более производительными и упрощает интеграцию с другими системами.

Далее мы обсудим некоторые другие сходства между GraphQL и REST.

## ****Архитектура****

И REST, и GraphQL реализуют несколько общих архитектурных принципов API. Например:

* Оба варианта не требуют сохранения состояния, поэтому сервер не сохраняет историю ответов между запросами.
* В обоих случаях используется модель «клиент-сервер», поэтому запросы от одного клиента вызывают ответы от одного сервера
* Оба протокола основаны на HTTP, поскольку HTTP является базовым протоколом связи

## ****Проектирование на основе ресурсов****

И REST, и GraphQL проектируют обмен данными на основе ресурсов. Ресурс – это любые данные или объект, к которым клиент может обращаться и манипулировать ими с помощью API. Каждый ресурс имеет свой уникальный идентификатор (URI) и набор операций (HTTP-методы), которые клиент может выполнять с ним.

Например, рассмотрим API социальных сетей, где пользователи создают публикации и управляют ими. В API, основанном на ресурсах, публикация будет ресурсом. Он имеет свой уникальный идентификатор, например /posts/1234. Кроме того, в нем есть набор операций, таких как запрос GET для получения сообщения в REST или запрос для получения сообщения в GraphQL.

## ****Обмен данными****

И REST, и GraphQL поддерживают схожие форматы данных.

JSON – самый популярный формат обмена данными, понятный всем языкам, платформам и системам. Сервер возвращает клиенту данные JSON. Другие форматы данных доступны, но используются реже, например XML и HTML.

Аналогичным образом, REST и GraphQL поддерживают кэширование. Таким образом, клиенты и серверы могут кэшировать часто используемые данные для повышения скорости связи.

## ****Нейтральность языка и базы данных****

API GraphQL и REST работают с любой структурой базы данных и любым языком программирования как на стороне клиента, так и на стороне сервера. Благодаря этому они хорошо совместимы с любыми приложениями.

## Какие ограничения в REST стремится устранить GraphQL?

Технология GraphQL возникла в 2012 году как ответ на потребность в скорости работы развивающихся платформ социальных сетей. Разработчики обнаружили, что существующие архитектуры API, такие как REST, слишком длинные и структурированные для эффективного создания новостных лент.

Далее мы обсудим некоторые проблемы, с которыми они столкнулись.

### ****Обмен данными с фиксированной структурой****

Для API REST требуется, чтобы запросы клиентов получали ресурс в фиксированной структуре. Эта жесткая структура проста в использовании, но она не всегда является наиболее эффективным средством для обмена именно теми данными, которые необходимы.

### ****Чрезмерная и недостаточная выборка****

API REST всегда возвращают весь набор данных. Например, от объекта человек в API REST вы получите имя, дату рождения, адрес и номер телефона этого человека. Вы получите все эти данные, даже если вам нужен только номер телефона.

Точно так же, если вы хотите узнать номер телефона человека и последнюю покупку, вам потребуется несколько запросов API REST. URL-адрес /person вернет номер телефона, а URL-адрес /purchase – историю покупок.

Разработчикам социальных сетей приходилось писать много кода только для обработки запросов API, что влияло на производительность и удобство работы пользователей.

Сервис GraphQL появился как решение на основе запросов. Запросы могут возвращать точные данные только за один запрос API и обмен ответами.

## Ключевые отличия: GraphQL и REST

API REST – это архитектурная концепция обмена данными между приложениями. С другой стороны, GraphQL – это спецификация, язык запросов API и набор инструментов. GraphQL работает на одном адресе с использованием HTTP.

Кроме того, при разработке REST больше внимания уделялось созданию новых API. В то же время разработчики GraphQL сосредоточились на производительности и гибкости API.

Далее приведены некоторые другие различия.

### ****Запрос на стороне клиента****

Ниже описано, как работает запрос REST.

* Глаголы HTTP, определяющие действие
* URL-адрес, определяющий ресурс, на котором следует выполнить команду HTTP
* Параметры и значения для анализа, если вы хотите создать или изменить объект в существующем серверном ресурсе

Например, вы используете запрос GET для получения из ресурса данных, доступных только для чтения, POST для добавления новой записи ресурса или PUT для обновления ресурса.

В отличие от этого, в запросах GraphQL используется следующее:

* Запрос на получение данных, доступных только для чтения
* Функция модификации данных
* Подписка на получение обновлений данных на основе событий или потокового вещания

Формат данных описывает, как сервер должен возвращать данные, включая объекты и поля, соответствующие схеме на стороне сервера. Можно также ввести новые данные. На внутреннем уровне GraphQL отправляет каждый запрос клиента в виде HTTP-запроса POST.

### ****Данные, возвращенные клиенту****

В соответствии с архитектурой REST данные возвращаются клиенту с сервера в соответствии со всей структурой ресурсов, указанной сервером. В следующих примерах показаны возвращаемые данные в REST и GraphQL.

**Пример возвращаемых данных в REST**

В REST запрос GET /posts возвращает следующее:

**«»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»**

[

  {

    "id": 1,

    "title": "First Post",

    "content": "This is the content of the first post."

  },

  {

    "id": 2,

    "title": "Second Post",

    "content": "This is the content of the second post."

  },

  {

    "id": 3,

    "title": "Third Post",

    "content": "This is the content of the third post."

  }

]

«»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»

**Пример возвращаемых данных в GraphQL**

При использовании GraphQL возвращаются только данные, указанные в структуре, заданной клиентом.

Запрос GET /graphql?query{post(id: 1) {id title content}} возвращает только первое сообщение:

**«»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»**

{

  "data": {

    "posts": [

      {

        "id": "1",

        "title": "First Post",

        "content": "This is the content of the first post."

      },

]}}

**«»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»»**

### ****Схема на стороне сервера****

GraphQL использует схему на стороне сервера для определения данных и сервисов данных, которая отличается от API REST.

Схема, написанная на языке определения схемы GraphQL, содержит следующие сведения:

* Типы объектов и поля, принадлежащие каждому объекту
* Функции преобразователя на стороне сервера, определяющие операцию для каждого поля

В схеме четко определены типы, описывающие все данные, доступные в системе, и способы доступа клиентов к этим данным или их изменения.

С другой стороны, для API REST схема на стороне сервера не требуется. Но вы можете определить ее дополнительно для эффективного проектирования API, документации и развития клиентов.

### ****Управление версиями****

По мере развития API их структуры данных и операции могут меняться. Клиенты, не знающие об этих изменениях, могут вывести из строя свои системы или допустить неизвестные ошибки.

Для решения этой проблемы в API REST часто используется функция определения версий в URL, например https://example.com/api/v1/person/12341. Однако управление версиями не является обязательным и может стать причиной ошибок.

Для GraphQL требуется обратная совместимость API. Таким образом, удаленные поля возвращают сообщение об ошибке, а поля с устаревшим тегом возвращают предупреждение.

### ****Обработка ошибок****

GraphQL – это строго типизированная архитектура API, поэтому она требует подробного описания данных, их структуры и операций с данными в схеме. Благодаря уровню детализации схемы система может автоматически выявлять ошибки запроса и предоставлять полезные сообщения об ошибках.

API REST слабо типизированы и вы должны встроить обработку ошибок в соответствующий код. Например, если запрос PUT анализирует числовое значение как текст, а не как целое число, система автоматически не идентифицирует ошибку.

## Когда использовать GraphQL, а когда – REST

Вы можете использовать API GraphQL и REST как взаимозаменяемые системы. Однако есть некоторые случаи использования, когда тот или иной вариант подходит лучше.

Например, если вы руководствуетесь этими требованиями, GraphQL, скорее всего, будет более подходящим выбором:

* У вас ограниченная пропускная способность и вы хотите свести к минимуму количество запросов и ответов
* У вас есть несколько источников данных и вы хотите объединить их в одном адресе
* У вас есть запросы клиентов, которые значительно различаются, и вы ожидаете совершенно разных ответов

С другой стороны, если вы руководствуетесь этими требованиями, REST, скорее всего, будет более подходящим выбором:

* У вас есть небольшие приложения с менее сложными данными
* У вас есть данные и операции, которые все клиенты используют одинаково
* У вас нет требований к сложным запросам данных



[Ссылка на начало](#_top)

# [Обзор серверных фреймворков: JavaScript, Python, C#](#_top)

## Фреймворки, библиотеки и прочие инструменты (из наиболее популярных)

* + **JavaScript**
  + Node.js

***Node.js*** *-*событийно-ориентированная платформа, применяемая для создания веб-приложений, которая обеспечивает возможность использовать JavaScript на стороне сервера.

Так как до появления Node.js язык JavaScript использовался только на стороне клиента, появление такой платформы было встречено разработчиками с энтузиазмом. По сути, это открывало новые  возможности для создания приложений с высокой производительностью и масштабируемостью.

 Нужно признать, что ожидания разработчиков оправдались, и на данный момент Node.js сохраняет популярность и продолжает двигаться вперед.

Сообщество Node.js растет и развивается. Регулярно генерируются новые идеи, а в результате появляются новые инструменты и библиотеки.

Благодаря таким темпам развития разработчики получили в свое распоряжение широкий ассортимент фреймворков, а подобное многообразие, как правило, предполагает сложный выбор.

## Что делает Node.js особенным?

* у этой платформы чистая и лаконичная кодовая база;
* она лучше всего подходит для гибкой разработки и прототипирования;
* имеет обширную экосистему, которая предоставляет библиотеки с открытым исходным кодом;
* может быть использована для быстрой разработки масштабируемых корпоративных приложений;
* написана на самом распространенном языке программирования – JavaScript.

**Несколько фактов о Node**

* Amazon, Netflix, LinkedIn, eBay, PayPal и Reddit используют Node.js для бэкэнда;
* 43% разработчиков применяют эту платформу для создания корпоративных приложений;
* в Paypal отметили заметное снижение среднего времени отклика на 35% после перехода на Node.js;
* Node.js может улучшить производительность любого существующего приложения на 50%;
* она снижает затраты на разработку приложений на 58%;
* Statista отметила, что Node.js является наиболее часто используемой разработчиками платформой по состоянию на начало 2020 года – ее доля составляет 51,4%.

## Преимущества использования фреймворков Node.js

* легкая масштабируемость;
* низкая кривая обучения;
* преимущества Fullstack JavaScript;
* большое и активное сообщество;
* использование единой кодовой базы для повышения общей производительности веб-сайта.

## Топ фреймворков Node.js

### Napi.js

**Hapi.js может быть использован для**:

* веб-сайтов;
* HTTP прокси;
* серверы API.

**Основные возможности Hapi.js:**

* проверка входных данных;
* регистрация;
* обработка ошибок;
* повторное использование кода;
* кэширование;
* отсутствие внешних зависимостей;
* функциональность на основе конфигурации;
* комплексная поддержка API аутентификации и авторизации.

**Когда использовать Hapi.js:**

Hapi.js идеально подходит для разработки безопасных, real-time, масштабируемых и социальных медиа-приложений. Большинство разработчиков мобильных приложений предпочитают Hapi.js для создания прокси-серверов и API-серверов.

### Nest.js

**Nest.js может быть использован для:**

* написания чистого и повторно используемого кода;
* написания кода с помощью конструкций высокого уровня: интерцепторы, фильтры, пайпы;
* написания масштабируемых и тестируемых приложений.

**Основные характеристики Nest.js:**

* легко расширяется – может использоваться с другими библиотеками;
* позволяет разработчикам использовать чистый JS;
* сочетает в себе черты функционального программирования, ООП и функционального реактивного программирования;
* предоставляет API фреймворков, которые помогают разработчику юзать различные сторонние модули, доступные для любой платформы;
* имеет подробную и поддерживаемую документацию.

**Когда использовать Nest.js:**

Nest.js используется для написания масштабируемых, тестируемых и слабо связанных приложений. Он обеспечивает правильный баланс структуры и гибкости, чтобы эффективно управлять кодом в крупных проектах.

### Express.js

Для этого фреймворка существует большое количество подробных инструкции и описаний, которые составлены разработчиками, проверившими его эффективность на практике. Поэтому именно с *Express* рекомендуется начинать работу, если вы намерены научиться создавать приложения на платформе Node.js.

Основная особенность этого фреймворка заключается в том, что для *Express* характерен **небольшой объем базового функционала**. Все остальные нужные вам функции нужно будет добирать за счет внешних модулей. По сути, *Express* в чистом виде – это сервер и у него может не быть ни одного модуля.

Благодаря такому минимализму разработчик изначально получает в свое распоряжение **легкий** **и быстрый инструмент**, который он может расширять и развивать.

При этом немаловажно, что выбор модулей для *Express* не связан ни с какими ограничениями: ни с количественными, ни с функциональными.

В результате, этот фреймворк обеспечивает разработчику возможность решать любые задачи, не ограничивая его при этом в выборе средств.

С одной стороны, не может не радовать тот факт, что отсутствие готовых универсальных решений  фактически означает, что каждое **создаваемое приложение будет уникальным.**

С другой стороны, разработчику нужно самостоятельно отбирать и организовывать модули, а это предполагает большой объем работы и соответственно, требует от разработчика больше времени и усилий.

**ПЛЮСЫ:**

✓ простота

✓ гибкость

✓ хорошая масштабируемость

✓ развитое сообщество

✓ подробная документация

✓ широкий выбор подключаемых модулей

**МИНУСЫ:**

✗ большой объем ручной работы

✗ используется устаревший подход callbacks функций

*а также koa.js и sails.js ну я не думаю что ему настолько дохуя надо. я устал. господи помогите.*

***Артём****: Да, брат, я тебя понимаю. Куча говна, а куда оно надо?*

*а еще я нихуя не понял кто что. типо интернет пишет что ехпресс это фреймворк для ноде.жс, но в записях лекций почему-то написано так будто первый это что-то отдельное стоящее столбняком. или так и есть. нихуя не понятно. вот какого хуя чуть ниже (чуть-чуть) ехпрес в отдельном пункте? а впрочем хуй с ним*

* + Express

## Java

### Spring

[Spring MVC](https://spring.io/) — один из самых популярных каркасов для web-сайтов на данный момент.

Поэтому он обладает развитой экосистемой — если вам не хватает чего-то при разработке, вы легко сможете найти дополнение для фреймворка, которое добавит необходимую возможность. Огромное сообщество разработчиков всегда будет готово прийти вам на помощь, если у вас возникнут вопросы. С помощью Spring MVC вы сможете без особого труда разрабатывать большие и быстроразвивающиеся сайты. Тем не менее, есть у него и несколько минусов: данный фреймворк довольно сложен для изучения, поэтому, несмотря на [хорошую документацию](https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/), новички, скорее всего, испытают трудности в его освоении. Кроме того, он не предоставляет удобных инструментов для создания пользовательского интерфейса.

• Spring Framework может быть задействован на всех архитектурных слоях, применяемых при при разработке web-приложений  
  
• Позволяет свободно связывать модули и легко их тестировать  
  
• Поддерживает декларативное программирование  
  
• Избавляет от самостоятельного создания фабричных и синглтон-классов  
  
• Поддерживает различные способы конфигурации

Несмотря на наличие стольких преимуществ, которыми обладает Spring, длительная процедура подготовки, связанная с его настройкой, способствовала появлению Spring Boot.

Spring обеспечивает слабо связанное приложение — это отличная функция. Однако, когда есть несколько слабо связанных блоков, отслеживание их становится утомительной и неблагодарной задачей.

### Vaadin

[Vaadin](https://vaadin.com/home) (произносится «вáадин») базируется на [Google Web Toolkit](http://www.gwtproject.org/" \o "http://www.gwtproject.org/) (он используется для вывода элементов пользовательского интерфейса и взаимодействия с сервером на стороне клиента), что добавляет архитектуре этого фреймворка сложности. Тем не менее, знание Google Web Toolkit не является обязательным для разработки на Vaadin. Более того, этот фреймворк вообще один из самых простых для изучения, а главной его «фичей» является «режим дизайна» —  в нем можно строить пользовательский интерфейс в режиме WYSIWIG и писать логику отдельно для каждого компонента. Документацию стоит упомянуть отдельно. Кроме [множества гайдов и демонстрационных примеров](https://vaadin.com/documentation) существует так называемая [«Книга Vaadin»](https://vaadin.com/book) — полный справочник по фреймворку, в котором вы найдете ответы на все свои вопросы. Книга бесплатно доступна онлайн, а бумажную версию можно заказать или получить на одной из многочисленных конференций, спонсируемых разработчиками фреймворка. Vaadin идеально подходит как для новичков, так и для профессионалов. Из минусов: не самая развитая экосистема и плохая масштабируемость для больших сайтов.

Одним из главных преимуществ использования фреймворка Vaadin является возможность использовать только Java, избегая прикруток языков веба (HTML, JS, XML). Да, это не единственный фреймворк, обладающий такой возможностью, но это пожалуй наименее проблемный из всех.  
Но есть и обратная сторона. Во-первых, Vaadin — это надстройка на GWT, что в конечном счете обяжет вас изучить и этот фреймворк. Во-вторых, как следует из всего описанного выше, он медлителен. Пользователь по сути работает лишь с представлениями, поэтому время выполнения любого действия можно смело умножать на 3 от номинального.

### **GWT**

*(почему кругляшки гуляют как бухие? они долбаебы? присмотрись - че за пиздец слева?? хддд. это я еще подравнял прикинь че было до этого)*

[GWT](http://www.gwtproject.org/) (Google Web Toolkit)

Vaadin и GWT чрезвычайно похожи и всячески связаны, но последнего есть одно важное отличие — он компилируется в JavaScript, с которым сервер и работает. Писать вам здесь тоже придется на чистом и немного ограниченном Java, но в отличие от Vaadin здесь больше самостоятельности в обработке запросов. С одной стороны, это позволяет увеличить быстродействие, наладить работу с памятью, с другой — потребует от вас больше знаний и навыков в разработке.

В остальном одни плюсы: множество API, виджетов, первоклассная поддержка, отсутствие необходимости безупречно знать Java — вполне достаточно, чтобы заинтересоваться данным фреймворком.

### JSF (JavaServer Faces) Тут нихуя нет

## Python

### ****Django****

***что я знал о джанго? Что я знал о джанго? Освобождённый нигер из восточного вутанга. Я клацающая клавиши, смешная обезьянка, Джанго это я, а я – это джанго***

**Django** — это бесплатная среда разработки веб-приложений Python с открытым исходным кодом, которая следует шаблону **Model-Template-View (MTV)**. Он был создан осенью 2003 года Адрианом Головатым и Саймоном Уиллисоном. **Django** был создан для упрощения процесса разработки сайта. Основное внимание уделяется повторно используемым компонентам, меньшему количеству кода и быстрой разработке.

### Flask

**Flask**— это микрофреймворк с удивительным началом. Это фактически началось, как первоапрельская шутка. Перед разработкой **Flask** Армин Роначер, создатель Flask, написал два других решения:

* **Werkzeug (серверная среда)** и
* **Jinja2 (библиотека шаблонов)**.

Армин подумал, что было бы интересно взять эти два решения и собрать их в zip-файл, поэтому он написал **Denied Framework** (как он называл его прежде, и назвал его Flask). Когда разработчик устанавливает этот Denied Framework, программа установки автоматически разархивирует файл и запускает эти два решения одновременно. Хотите верьте, хотите нет, но Flask получил положительную реакцию.

**Основные различия между Django и Flask**

**Django** предоставляет свой собственный **Django ORM** (объектно-реляционное отображение) и использует модели данных, в то время как **Flask** вообще не имеет моделей данных. Модели данных позволяют разработчикам связывать таблицы базы данных с классами на языке программирования, чтобы они могли работать с моделями так же, как ссылки на базы данных. Почему у Flask нет модели данных? Потому что философия Flask отличается от философии Django.

***Django связывает все вместе, а Flask более модульный.***

Основное различие между Django и Flask в том, что **Django предоставляет полнофункциональную среду Model-View-Controller.**

**Цель Django** — упростить процесс разработки сайта. Он основан на меньшем количестве кода, повторно используемых компонентах и ​​быстрой разработке.

**Flask**, с другой стороны, представляет собой **микрофреймворк, основанный на концепции «хорошо выполнять одну вещь»**. Он не предоставляет ORM и поставляется только с базовым набором инструментов для веб-разработки.

Приложения для **Flask** в основном представляют собой одностраничные приложения (SPA). Это хороший выбор для небольших и средних сайтов, таких как форумы и личные блоги. **Django** идеально подходит для крупных проектов, таких как сайты электронной коммерции и CMS.

Преимущества и недостатки этих каркасов

**Производительность**

Если учесть, что производительность **Django vs Flask** имеет хорошие результаты и используется на сайтах с большим трафиком, что является отличным показателем их эффективности.

**Пакеты**

**Flask** минималистичен и не имеет ограничений, то есть разработчики могут реализовать именно то, что они хотят, используя внешние библиотеки. Это делает **Flask** гибким и расширяемым.

**Django**, с другой стороны, имеет огромное количество встроенных пакетов. Если быть точным, то на сентябрь 2019 года было 4046 пакетов Django. Это означает, что вы, скорее всего, найдете пакет для сборки и запуска своего приложения с меньшими усилиями.

**Сообщество**

В случае сравнения **Python Django с Flask**, Django имеет огромное и активное сообщество разработчиков. Если у вас есть какие-либо вопросы, вы можете задать их на различных веб-порталах и форумах, таких как **Stack Overflow**, и, скорее всего, получите ответ. Кроме того, не трудно найти работу Django.

Сообщество Flask не такое большое, как сообщество Django.

**Случаи использования**

**Django** был разработан для быстрой разработки сложных веб-приложений. Он предоставляет разработчикам необходимые инструменты для реализации масштабируемой и поддерживаемой функциональности. С другой стороны, простота Flask позволяет разработчикам быстрее создавать небольшие приложения.

**Выберите Django,** если у вас мало времени или у вас ограниченный бюджет. Django — это идеальное решение для MVP, платформ электронной коммерции, приложений mHealth и правительственных веб-сайтов.

**Выберите Flask,** если у вас небольшой или средний проект, который имеет уникальные требования и требует пользовательских компонентов.

## C#

### ASP.NET MVC

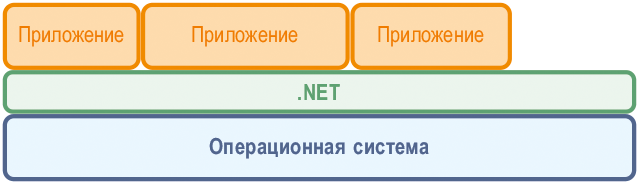
### ASP.NET

* **ASP.NET (Active Server Pages для .NET)** — технология создания веб-приложений и веб-сервисов от компании Майкрософт. Она является составной частью платформы Microsoft .NET и развитием более старой технологии Microsoft ASP. На данный момент последней версией этой технологии является ASP.NET 5.
* **Платформа ASP.NET MVC** представляет собой фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна MVC.
* **Платформа Web API** представляет иной способ построения приложения ASP.NET несколько отличный от ASP.NET MVC. Web API представляет собой веб-службу, которая может взаимодействовать с различными приложениями. При этом приложение может быть веб-приложением ASP.NET, либо может быть мобильным или обычным десктопным приложением.
* **Платформа Web API** не является частью фреймворка ASP.NET MVC и может быть задействована как в связке с MVC, так и в соединении с Web Forms. Поэтому в Web API имеется своя система версий. Так, первая версия появилась с .net 4.5. А вместе с .NET 4.5.1 и MVC 5 вышла Web API 2.0.
* **Платформа ASP.NET Core** представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов.
* **Платформа ASP.NET Core** построен на основе кросс-платформенной среды .NET Core, которая может быть развернута на основных популярных операционных системах: Windows, Mac OS X, Linux. И хотя Windows в качестве среды для разработки и развертывания приложения до сих пор превалирует, но теперь уже мы не ограничены только этой операционной системой. То есть мы можем запускать веб-приложения не только на ОС Windows, но и на Linux и Mac OS. А для развертывания веб-приложения можно использовать традиционный IIS, либо кросс-платформенный веб-сервер Kestrel.

.**NET Core** — это горизонтальное развитие программной платформы .NET в другие операционные системы, в которых она обеспечивает возможность использования приложений, разработанных для Windows.

Несколько слов о .NET

* Одна из существенных причин головной боли программистов — непредсказуемость условий, в которых будут запускаться написанные ими приложения. — Разная конфигурация аппаратной части, разные версии операционных систем, разные наборы библиотек, разные устройства, разные драйверы, …
* Попытки как-то унифицировать среду выполнения приложений предпринимались, предпринимаются и будут предприниматься.
* Мощной и довольно успешной попыткой унификации программной среды в операционной системе Windows является платформа [.NET](https://www.microsoft.com/net) («дот нет»), разработанная компанией Microsoft в начале 2000-х годов.
* Основная идея состояла в том, чтобы между операционной системой и приложением поместить программную прослойку, которая сглаживала бы особенности конкретного экземпляра операционной системы, обеспечивая приложению «идеальную» среду исполнения.
* В рамках статьи мы не будем углубляться в технические детали этой прослойки. Желающие могут обратиться к её [технической документации](https://docs.microsoft.com/ru-ru/dotnet/).
* Очень упрощённо устройство .NET можно проиллюстрировать следующим рисунком.

ОТКРЫТОСТЬ

* Как уже было сказано, платформа .NET является разработкой и собственностью компании Microsoft, но была и остаётся бесплатной в использовании.
* Программный код платформы был размещён в публичном репозитории [GitHub](https://github.com/DotNet" \o "https://github.com/DotNet) и теперь распространяется как продукт с открытым исходным кодом.

КРОССПЛАТФОРМЕННОСТЬ

* Сегодня на потребительском рынке наиболее распространены три: Windows, macOS и Linux (с вариациями).
* Разработчик удачного, востребованного пользователями приложения заинтересован в его распространении в разных операционных системах. Однако в большинстве случаев просто так запустить приложение, разработанное для другой операционной системы, невозможно — требуется его «портировать».
* В каких-то ситуациях можно просто перекомпилировать исходный текст программы для новой операционной системы. Но в большинстве случаев такой трюк не проходит, и требуется переписывать программу целиком или какие-то её части. Именно так пришлось действовать компании Microsoft, когда в середине 90-х она захотела сделать свой Office доступным в MacOS.
* Но адаптации исходного кода приложения можно избежать, если между ним и операционной системой будет находиться «стандартизирующая» прослойка. Например, … .NET. Тогда приложение, разработанное в одной операционной системе, можно будет без проблем запускать в другой. То есть вместо портирования отдельных приложений, можно портировать саму программную прослойку.
* Так Microsoft и поступила. Правда, портировала она не текущую версию платформы .NET, существующую в ОС Windows, а некую производную от неё, которую назвала [.NET Core](https://docs.microsoft.com/ru-ru/dotnet/core/).

В значительной степени её функциональность совпадает с функциональностью исходной .NET, но одновременно — в чём-то шире, а в чём-то уже. В целом, можно считать, что .NET и .NET Core совместимы друг с другом.

### ASP.NET Core

Несколько слов о .NET Core

Сейчас платформа .NET Core существует для трёх операционных систем: [Windows](https://www.microsoft.com/net/download/windows/), [Linux](https://www.microsoft.com/net/download/linux" \o "https://www.microsoft.com/net/download/linux) и [macOS](https://www.microsoft.com/net/download/macos" \o "https://www.microsoft.com/net/download/macos). То есть приложение, разработанное на базе .NET Core, может быть без изменения запущено во всех операционных системах, в которых имеется указанная платформа.

Не будем лукавить, идеальная 100-процентная совместимость трудно достижима. В настоящее время на платформе .NET Core не реализованы Windows Forms, Windows Presentation Foundation (WPF), WebForms. Это означает, что приложения с развитым графическим интерфейсом портироваться таким способом в настоящее время не могут. Однако в информационных системах существует много задач, решение которых не требует пользовательского графического интерфейса.

## P**hp**

**Зачем нужен PHP-фреймворк**

PHP — один из самых популярных и востребованных языков программирования. Его активно используют крупные проекты, например, Facebook и «ВКонтакте». На PHP написаны популярные системы управления контентом (CMS), в том числе WordPress. На этом движке работает около трети всех сайтов в интернете и около 60 % сайтов на CMS.

Мы поможем выбрать наиболее подходящий фреймворк для любого вашего проекта.

Чтобы найти правильный ответ на этот вопрос, нужно проанализировать такие пункты:

**Цель**

Основным фактором в процессе выбора фреймворка должно стать предназначение проекта. Например, для простого приложения нет необходимости использовать продукт промышленного уровня, который требует массу лишних ресурсов.

**Сообщество**

Чем выше популярность PHP-фреймворка, тем проще получить обратную связь от его создателей и найти дополнительные библиотеки или внешние пакеты. Имеет значение и ваше местоположение: в русскоговорящих странах предпочитают Yii, тогда как сообщество Laravel составляют по большей части англоговорящие разработчики.

**Безопасность**

Разрабатывая сайты на PHP, нельзя не уделять особенное внимание проблеме безопасности, ведь любой веб-сервис потенциально уязвим. Советуем всегда использовать такие фреймворки, которые в этом плане обладают продвинутыми современными функциями.

**Документация**

Благодаря подходящей подробной документации программистам легче ориентироваться в исходном коде, а значит, и заниматься самой разработкой. Фреймворки бывают разного уровня сложности: изучить Yii, к примеру, нетрудно, а вот для освоения Symfony придется приложить заметно больше усилий. Поэтому, прежде чем сделать выбор инструмента, обязательно оцените качество и количество имеющейся документации.

Топ-4 PHP-фреймворка:

* Symfony;
* Laravel;
* Yii 2;
* Zend;

Они дают возможность делать как образцовые веб-приложения, так и качественное корпоративное ПО.

## Фреймворки

### [Symfony](https://symfony.com/)

Фреймворк появился в 2005 году. Сейчас он имеет встроенную систему тестирования, позволяет взаимодействовать с компонентами, а также многократно применять код. К другим важным преимуществам Symfony относятся:

* **Быстрая загрузка.** Для потребителей производительность приложения стоит далеко не на последнем месте. Именно от нее зависит, будут ли вообще им пользоваться. В версии 4.2 загрузка REST API происходит за 2 мс, т.е. Symfony на сегодняшний день является самым скоростным PHP-фреймворком.
* **Подробная документация.** Детальная документация данного инструмента пригодится не только начинающим, но и опытным программистам. В ней понятно и с примерами описан каждый компонент фреймворка.
* **Гибкость.** Symfony полностью конфигурируем за счет Event Dispatcher, он способен приспособиться к самым разным требованиям проекта. Плюс ко всему, компоненты фреймворка управляются независимо один от другого.
* Надежность. Компания-разработчик занимается поддержкой Symfony уже больше 13 лет. Так что в случае возникновения проблем вам будет к кому обратиться.

### [Laravel](https://laravel.ru/)

Laravel изначально проектировался под создание многофункциональных веб-приложений. Он может похвастать отличным движком шаблонов, который выполняет множество обычных задач. Что касается функционала, нужно отметить Restful Routing, аутентификацию и кэширование. Благодаря этим опциям Laravel делает разработку более быстрой. Но есть и другие причины, по которым мы советуем применять этот фреймворк:

* **Высокий уровень безопасности.** Мы уже акцентировали внимание на том, что для PHP-фреймворка безопасность является очень важной характеристикой. В данном случае вы будете работать в безрисковой среде, потому как Laravel блокирует все вредоносные активности и обладает прекрасной защитой от случайных и скрытых внедрений SQL-кода.
* **Встроенные почтовые сервисы.** За счет интегрированного почтового сервиса разработанное вами веб-приложение сможет отправлять пользователям SMS и Slack уведомления. Фреймворк имеет простой в использовании API, а также драйвера для Mailgun, Amazon SES, SMTP и Mandrill.
* **Обширный функционал.** Разработчики могут внедрять разные виды готового функционала благодаря пакетам, плагинам, компонентам и модулям, что позволяет существенно сократить время разработки.
* Улучшенная аутентификация. Фреймворк позволяет пользователям без особых усилий создавать логику аутентификации. А это, как известно, далеко не самая простая задача.

### [Yii2](https://www.yiiframework.com/)

Высокая скорость загрузки и усиленная безопасность – вот чем известен Yii2. В этом инструменте также заложен принцип Don't repeat yourself, т.е. неповторения кода. А с помощью Ajax и JQuery фреймворк облегчает создание высокомасштабируемых веб-приложений. Достоинствами Yii2, помимо прочего, являются:

* **Улучшенная безопасность.**Yii2 способен защищать сайты и приложения от разного рода атак, в том числе подделки cookie, внедрения SQL-кода, CSRF и межсайтового скриптинга.
* **Ускоренная разработка.** Фреймворк генерирует базовые функции CRUD, поэтому программисты работают быстрее, что не только экономит время, но и уменьшает стоимость разработки.
* Простая установка. Когда каждая минута на счету, Yii радует пользователей шаблоном инсталляции. В результате установка и настройка продукта происходят в разы быстрее.

### [Zend Framework](https://framework.zend.com/)

Этот продукт создан по методологии agile и ориентирован на разработку корпоративных веб-приложений. Zend позволяет внедрять в проект любые специфичные фичи. Пятилетняя поддержка дает возможность делать стабильные бизнес-программы с длительным жизненным циклом.

Кроме безопасности и великолепной производительности Zend обладает такими преимуществами:

* **Набор компонентов аутентификации.** Процесс аутентификации усложняет PHP-разработку. Решением этой проблемы во фреймворке выступает коллекция инструментов для тестирования форм. Таким образом, этот функционал не приходится разрабатывать с нуля.
* **Zend Studio.** Это проприетарная интегрированная среда, в которой разработчики могут писать код намного быстрее. Отладка становится более удобной благодаря интеграции Xdebug, Z-Ray и Zend Debugger. Также есть возможность делать развертку на разных серверах, в частности поддерживается  Microsoft Azure и Amazon Web Services.
* **Улучшенная документация.** В руководстве по фреймворку описано более 500 примеров на одной тысяче страниц. Там вы точно сможете найти подходящие решения для своих проектов. В качестве бонуса Zendcasts содержит множество обучающих роликов, посвященных функционалу этого инструмента.

*Zend иди нахуй зенд.*

кстати прикиньте ебало чела который сидит это все ищет перекопирует подгоняет +- под один вид вставляет гиперссылки и прочую хуйню.так вот. это мое ебало

## Ruby.

### Ruby on Rails

**Для чего нужен Ruby on Rails: Советы от Back-end разработчиков**

**Что такое Ruby и где он используется?**

Итак, давайте начнем с того, что такое Ruby и для чего он используется. Основное различие между Ruby и Ruby on Rails скрыто в их предназначении. Ruby — это язык программирования, который был создан в 1990-х годах Юкихиро «Мац» Мацумото. Основная задача Ruby — быстрое создание новых проектов с высокой производительностью. Это язык программирования общего назначения, такой же, как C ++ или PHP. Язык программирования Ruby постоянно развивается и окружен большим сообществом.

Обширная поддержка сообщества, удобство для пользователя, простота, удобочитаемость, гибкость и жемчужины сообщества, которые являются скриптами кодирования, которые упрощают процесс разработки, являются основными преимуществами языка программирования Ruby, которые называют внутренние разработчики.

Для чего используется Ruby? На этот вопрос нет однозначного ответа, поскольку, как и любой язык программирования общего назначения, он подходит для широкого спектра задач программирования.

Однако создание нового программного обеспечения с использованием только Ruby — утомительная работа. Именно поэтому была создана специальная структура для оптимизации процесса разработки.

**Что такое Ruby on Rails?**

Rails часто называют главной причиной популярности Ruby. Rails, или Ruby on Rails, является средой с открытым исходным кодом, написанной на языке программирования Ruby и основанной в 2003 году Дэвидом Хайнемайером Ханссоном, который также известен под именем DHH

Платформа Ruby on Rails значительно упрощает процесс создания веб-сайтов и приложений, снимая общие повторяющиеся задачи с плеч разработчиков, такие как создание форм, таблиц и меню. Разработчикам не нужно создавать новый веб-сайт или веб-приложение с нуля, поскольку они могут использовать готовые решения для повторяющихся задач. Ниже мы поговорим подробнее о том, для чего используется Ruby on Rails в реальных приложениях.

**Почему Ruby on Rails так популярен?**

Статус Open Source — это первое, что нужно учитывать при выборе правильной серверной среды. Это означает, что Ruby on Rails является бесплатным и может использоваться бесплатно. Любой желающий может пойти и скачать исходный код для дальнейшего использования в своих проектах.

В Ruby on Rails огромное сообщество талантливых и опытных разработчиков. Сообщество разрабатывает множество бесплатных дополнений, которые можно интегрировать в приложения. Они очень полезны для стартапов, которые хотят запустить новое многофункциональное приложение в кратчайшие сроки.

**Преимущества Ruby on Rails Development**

Разработка на Ruby on Rails имеет ряд преимуществ для проектов:

* Обширная экосистема
* Ruby on Rails MVC
* Согласованность и чистый код
* DRY
* Высокая масштабируемость
* Безопасность
* Время и эффективность затрат
* RAD
* Самодокументирование
* Тестовая среда
* Соглашение по конфигурации

**Обширная экосистема**

Именно его экосистема делает Ruby on Rails превосходным по сравнению со многими другими фреймворками.

**Ruby on Rails MVC**

Ruby on Rails MVC допускает параллельную разработку и позволяет программистам ускорить процесс разработки в три раза.

**Согласованность и чистый код**

Разработчики Ruby on Rails могут использовать готовые к использованию части кода, что упрощает реализацию многих функций. В результате код приложения является чистым и имеет высокую читаемость.

**DRY**

DRY (не повторяй себя) — еще один принцип, на котором основан Ruby on Rails. Если есть повторяющаяся задача, то при разработке на Ruby on Rails вам не нужно писать один и тот же код снова и снова. Фреймворк воспринимает повторяющиеся задачи таким образом, что фоновые разработчики могут использовать их неограниченное количество раз.

**Высокая масштабируемость**

Масштабируемость Ruby on Rails — еще одно преимущество. Приложение, построенное на RoR, может быть масштабировано для обработки тысяч запросов в секунду, отправленных несколькими пользователями. Отличным примером высокой производительности Ruby on Rails является платформа электронной коммерции Shopify, которая обрабатывает до 80 000 запросов в секунду.

**Безопасность**

Ruby on Rails не рискует вопросами безопасности. Безопасность Ruby on Rails — еще одно преимущество. В инфраструктуру встроены некоторые ориентированные на безопасность функции, которые делают приложения защищенными от SQL-инъекций и XSS-атак. Кроме того, вы можете найти много драгоценных камней, которые направлены на другие угрозы безопасности.

**Время и эффективность затрат**

Время часто является основным препятствием для стартапов. Все перечисленные выше функции в совокупности делают Ruby on Rails экономичным по времени и затратам.

**RAD**

Быстрая разработка приложений (RAD) — еще одна сфера, для которой используется Ruby on Rails. Структура оптимизирует процесс изменения размещения.

**Самодокументирование**

Как мы уже упоминали, код Ruby хорошо читается и самодокументируется (самоописание). Это ускоряет процесс разработки, так как команде разработчиков не нужно выписывать отдельную документацию. У новичков в командах разработчиков не должно быть проблем с пониманием концепции и участием в существующих проектах.

**Тестовая среда**

В Ruby on Rails есть три среды по умолчанию, а именно: производство, разработка и тестирование. Весь цикл разработки оптимизирован, и вы можете протестировать продукт, который разрабатывается на каждом этапе. Это приводит к меньшему количеству ошибок и ошибок, о которых вы должны знать и отлаживать. Это важный фактор, который необходимо учитывать при попытке определить, для чего используется Ruby on Rails.

**Соглашение по конфигурации**

Соглашение о конфигурации является одним из ключевых принципов разработки Ruby on Rails. Это сокращает время, которое программисты тратят на настройку файлов. Платформа Ruby on Rails имеет набор правил, облегчающих начинающим разработчикам Ruby on Rails использование платформы. Благодаря соглашениям код становится читабельным и лаконичным и позволяет легко перемещаться по веб-приложению Ruby on Rails.

**Недостатки разработки Ruby on Rails**

Несмотря на то, что у Ruby on Rails есть много преимуществ, у него есть несколько недостатков, которые необходимо учитывать, прежде чем принимать решение о том, для чего используется Ruby on Rails и подходит ли он:

* Документация
* Скорость выполнения
* Скорость загрузки
* Хостинг сайтов

**Документация**

В среде Ruby on Rails документацию иногда сложно найти. Эта проблема особенно актуальна при использовании драгоценных камней, так как немногие разработчики склонны документировать все.

**Скорость выполнения**

Ruby и Rails работают быстро, но не так быстро, как подавляющее большинство других объектно-ориентированных языков программирования. Скорость выполнения часто называют главным аргументом против Ruby on Rails. По сравнению со скоростью исполнения Ruby on Rails против Node.JS против отстает. Тем не менее, когда мы рассматриваем Java-фреймворк Spring, RoR побеждает в этой битве. Вы вряд ли будете иметь узкие места в производительности Ruby on Rails.

**Скорость загрузки**

Разработчики часто называют скорость загрузки одним из разочаровывающих аспектов среды Ruby on Rails. В зависимости от количества используемых файлов и драгоценных камней запуск платформы может занять значительное время. Spring, предварительный загрузчик приложения Rails, решил эту проблему как-то; Тем не менее, еще предстоит проделать определенную работу, чтобы полностью решить проблему.

**Хостинг сайтов**

Не все хосты сайта поддерживают Ruby on Rails. Фреймворк требует гораздо больше ресурсов, чем PHP, и низкоуровневые хостинг-провайдеры не могут обеспечить необходимые вычислительные мощности. Однако вы можете найти множество провайдеров хостинга веб-сайтов, которые поддерживают приложения Ruby on Rails.

[Ссылка в начало)](#_top)

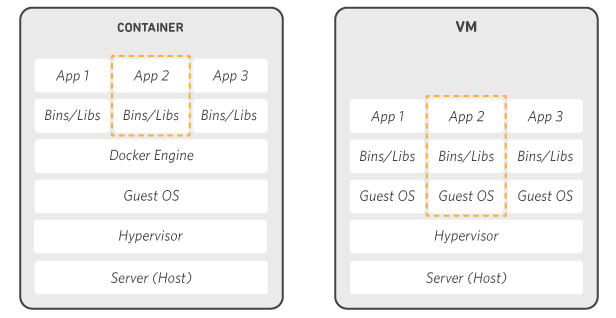
# [Doker. Цель технологии. Основные сведения. Термины и концепции. Файл Dockerfile.](#_top)

Docker – это программная платформа для быстрой разработки, тестирования и развертывания приложений. Docker упаковывает ПО в стандартизованные блоки, которые называются [контейнерами](https://aws.amazon.com/ru/containers/). Каждый контейнер включает все необходимое для работы приложения: библиотеки, системные инструменты, код и среду исполнения. Благодаря Docker можно быстро развертывать и масштабировать приложения в любой среде и сохранять уверенность в том, что код будет работать.

Использование Docker на AWS предоставляет разработчикам и системным администраторам надежный и экономичный способ сборки, доставки и запуска распределенных приложений любого масштаба.

## Как работает Docker

В основе работы Docker лежит стандартизированный способ исполнения кода. Docker – это операционная система для контейнеров. Подобно тому как [виртуальная машина](https://aws.amazon.com/ru/ec2/) создает виртуальное представление аппаратного обеспечения сервера (то есть устраняет необходимость непосредственно управлять таковым), контейнеры создают виртуальное представление серверной операционной системы. После установки на каждый сервер Docker предоставляет доступ к простым командам, необходимым для сборки, запуска или остановки контейнеров.



## **Почему следует использовать Docker**

Использование Docker позволяет быстрее и эффективнее доставлять или перемещать код, стандартизирует выполняемые приложениями операции и в целом экономит средства, оптимизируя использование ресурсов. Благодаря Docker пользователи получают объект, который с высокой надежностью можно запускать на любой платформе. Простой и понятный синтаксис Docker обеспечивает полный контроль над выполняемыми операциями. Повсеместное внедрение контейнеров подразумевает доступ к разнообразным инструментам и готовым приложениям, которые можно использовать с Docker.

**БЫСТРАЯ ДОСТАВКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

Пользователи Docker осуществляют доставку приложений в среднем в семь раз чаще, чем те, кто не использует Docker. Docker позволяет доставлять изолированные сервисы с необходимой периодичностью.

**СТАНДАРТИЗАЦИЯ ОПЕРАЦИЙ**

Доставка приложений в небольших контейнерах упрощает процесс развертывания, обнаружения проблем и отката для их устранения к прошлым версиям.

**ЭФФЕКТИВНОЕ ПЕРЕМЕЩЕНИЕ**

Приложения на основе контейнеров Docker можно эффективно переносить с локальных машин, на которых ведется разработка, в AWS для рабочего развертывания.

**ЭКОНОМИЯ**

Контейнеры Docker позволяют исполнять больше кода на каждом сервере, повышая эффективность использования ресурсов и сокращая расходы.

**Когда рекомендуется использовать Docker**

Контейнеры Docker можно использовать в качестве основных компонентов для создания современных платформ и приложений. Docker упрощает сборку и запуск распределенных микросервисных архитектур, развертывание кода с помощью стандартизированных конвейеров непрерывной интеграции и доставки, создание высокомасштабируемых систем обработки данных и полностью управляемых платформ для разработчиков. Сотрудничество AWS и Docker позволит упростить операцию развертывания артефактов Docker Compose в Amazon ECS и AWS Fargate.

**МИКРОСЕРВИСЫ**

Воспользуйтесь преимуществами стандартизированного развертывания кода с помощью контейнеров Docker для создания и масштабирования распределенных архитектур приложений.

**НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ И ДОСТАВКА**

Ускорьте доставку приложений за счет стандартизации рабочих сред и устранения расхождений между стеками и версиями языков программирования.

**ОБРАБОТКА ДАННЫХ**

Обработки больших данных как сервис. Объедините пакеты данных и аналитики в портативные контейнеры, запуск которых не требует специальных технических навыков.

**КОНТЕЙНЕР КАК СЕРВИС**

Создавайте и доставляйте распределенные приложения, содержание и инфраструктура которых программно управляема и защищена.

*блядские термины:*

*если буквально в двух словах:*

Механизмы Docker:

1. [Платформа Docker](#_▍Платформа_Docker) — ПО, благодаря которому можно работать с контейнерами.
2. [Движок Docker](#_▍Движок_Docker) — клиент-серверное приложение (CE или Enterprise).
3. [Клиент Docker](#_▍Клиент_Docker) — программа, которая позволяет взаимодействовать с демоном Docker посредством CLI.
4. [Демон Docker](#_▍Демон_Docker) — сервер Docker, отвечающий за управление ключевыми механизмами системы.
5. [Тома Docker](#_▍Тома_Docker) — хранилище информации, используемое в контейнерах.
6. [Реестр Docker](#_▍Реестр_Docker) — удалённое хранилище образов.
7. [Хаб Docker](#_▍Хаб_Docker) — самый крупный реестр Docker, используемый по умолчанию.
8. [Репозиторий](#_▍Репозиторий_Docker) — коллекция образов Docker с одним и тем же именем.

Масштабирование:

1. [Сетевая подсистема Docker](#_▍Сеть_Docker) — среда, которая позволяет организовывать взаимодействие контейнеров.
2. [Docker Compose](#_▍Docker_Compose) — технология, упрощающая работу с многоконтейнерными приложениями.
3. [Docker Swarm](#_▍Docker_Swarm) — средство для управления развёртыванием контейнеров.
4. [Сервисы Docker](#_▍Сервисы_Docker) — контейнеры в продакшне.

Если с умными словами:

## Механизмы Docker

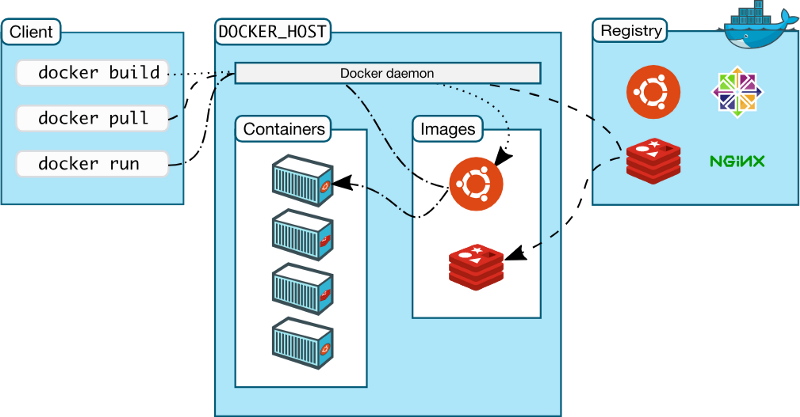
### Платформа Docker

Платформа Docker ([Docker Platform](https://docs.docker.com/engine/docker-overview/" \l "the-docker-platform)) — это программа, которая даёт нам возможность упаковывать приложения в контейнеры и запускать их на серверах. Платформа Docker позволяет помещать в контейнеры код и его зависимости. Как результат, системы, основанные на контейнерах, легко масштабировать, так как контейнеры можно переносить и воспроизводить.

### Движок Docker

Движок Docker ([Docker Engine](https://www.docker.com/products/docker-engine)) — это клиент-серверное приложение. Компания Docker разделила движок Docker на два продукта. [Docker Community Edition](https://docs.docker.com/install/) (CE) — это бесплатное ПО, во многом основанное на [опенсорсных инструментах](https://opensource.stackexchange.com/questions/5436/is-docker-still-free-and-open-source).  
  
Вероятно, вы будете пользоваться именно этой версией Docker. [Docker Enterprise](https://www.docker.com/products/docker-enterprise) — это платная версия системы, дающая пользователям дополнительные возможности в области поддержки систем, управления ими и безопасности. Платная версия Docker даёт компании средства, необходимые для её существования.

### Клиент Docker



*Клиент Docker и другие механизмы экосистемы (взято из*[*документации*](https://docs.docker.com/engine/docker-overview/)*)*  
  
Клиент Docker ([Docker Client](https://docs.docker.com/engine/docker-overview/)) — это основное средство, которое используют для взаимодействия с Docker. Так, при работе с интерфейсом командной строки Docker ([Docker Command Line Interface](https://docs.docker.com/engine/reference/commandline/cli/), CLI), в терминал вводят команды, начинающиеся с ключевого слова docker, обращаясь к клиенту. Затем клиент использует API Docker для отправки команд демону Docker.

### Демон Docker

Демон Docker ([Docker Daemon](https://docs.docker.com/engine/docker-overview/)) — это сервер Docker, который ожидает запросов к API Docker. Демон Docker управляет образами, контейнерами, сетями и томами.

### Тома Docker

Тома Docker ([Docker Volumes](https://docs.docker.com/storage/volumes/)) представляют собой наиболее предпочтительный механизм постоянного хранения данных, потребляемых или производимых приложениями.

### Реестр Docker

Реестр Docker ([Docker Registry](https://hub.docker.com/)) представляет собой удалённую платформу, используемую для хранения образов Docker. В ходе работы с Docker образы отправляют в реестр и загружают из него. Подобный реестр может быть организован тем, кто пользуется Docker. Кроме того, поставщики облачных услуг могут поддерживать и собственные реестры. Например, это касается [AWS](https://aws.amazon.com/ecr/) и [Google Cloud](https://cloud.google.com/container-registry/).

### Репозиторий Docker

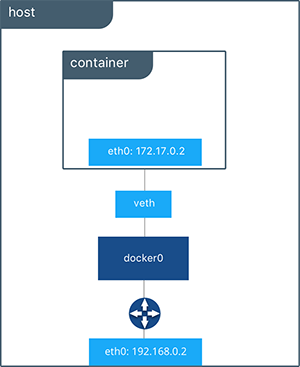
Репозиторием Docker ([Docker Repository](https://docs.docker.com/docker-hub/repos/)) называют набор образов Docker, обладающих одинаковыми именами и разными тегами. Теги — это идентификаторы образов.  
  
Обычно в репозиториях хранятся разные версии одних и тех же образов. Например, [Python](https://hub.docker.com/_/python) — это имя популярнейшего официального репозитория Docker на хабе Docker. А вот Python:3.7-slim — это версия образа с тегом 3.7-slim в репозитории Python. В реестр можно отправить как целый репозиторий, так и отдельный образ.  
  
Масштабирование решений, основанных на контейнерах

Следующие четыре термина имеют отношение к одновременному использованию нескольких контейнеров.

### Хаб Docker

Хаб Docker ([Docker Hub](https://hub.docker.com/)) — это самый крупный реестр образов Docker. Кроме того, именно этот реестр используется при работе с Docker по умолчанию. Пользоваться хабом Docker можно бесплатно.

### Сеть Docker



*Сеть Docker (взято из*[*документации*](https://docs.docker.com/engine/tutorials/networkingcontainers/)*)*  
  
Сетевые механизмы Docker ([Docker Networking](https://docs.docker.com/engine/tutorials/networkingcontainers/)) позволяют организовывать связь между контейнерами Docker. Соединённые с помощью сети контейнеры могут выполняться на одном и том же хосте или на разных хостах. Подробности о сетевой подсистеме Docker можно почитать [здесь](https://www.oreilly.com/learning/what-is-docker-networking).

### Docker Compose

[Docker Compose](https://docs.docker.com/compose/) — это инструмент, который упрощает развёртывание приложений, для работы которых требуется несколько контейнеров Docker. Docker Compose позволяет выполнять команды, описываемые в файле docker-compose.yml. Эти команды можно выполнять столько раз, сколько потребуется. Интерфейс командной строки Docker Compose упрощает взаимодействие с многоконтейнерными приложениями. Этот инструмент устанавливается при установке Docker.

### Docker Swarm

[Docker Swarm](https://docs.docker.com/engine/swarm/) — это решение, предназначенное для управления контейнерными развёртываниями (то есть, как говорят, для оркестрации контейнеров). В [этом](https://docs.docker.com/get-started/#recap-and-cheat-sheet) материале из официального учебного курса по Docker можно найти сведения о Docker Swarm. Мне хотелось бы порекомендовать вам не тратить время на изучение Docker Swarm в том случае, если у вас нет на то веской причины.

### Сервисы Docker

Сервисы Docker ([Docker Services](https://docs.docker.com/get-started/part3/" \l "introduction)) — это различные части распределённого приложения. Вот что о них говорится в [документации](https://docs.docker.com/get-started/part3/#introduction):  
  
*Сервисы — это всего лишь «контейнеры в продакшне». В пределах сервиса выполняется лишь один образ, но сервис определяет то, как именно выполняется образ. В частности, речь идёт о том, какие порты должны использоваться, сколько реплик контейнера должно выполняться для того, чтобы сервис обеспечивал бы необходимую вычислительную мощность, и так далее. Масштабирование сервисов предусматривает изменение количества экземпляров контейнера, в которых работает некая программа, благодаря чему сервису выделяется столько системных ресурсов, сколько ему требуется для решения некоей задачи.*  
  
Сервисы Docker позволяют масштабировать контейнеры в пределах нескольких демонов Docker, благодаря им существует и технология Docker Swarm.

*Файлы докер ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))*

### Образы Docker

Вспомните о том, что контейнер Docker — это образ Docker, вызванный к жизни. Это — самодостаточная операционная система, в которой имеется только самое необходимое и код приложения.  
  
Образы Docker являются результатом процесса их сборки, а контейнеры Docker — это выполняющиеся образы. В самом сердце Docker находятся файлы Dockerfile. Подобные файлы сообщают Docker о том, как собирать образы, на основе которых создаются контейнеры.  
  
Каждому образу Docker соответствует файл, который называется Dockerfile. Его имя записывается именно так — без расширения. При запуске команды docker build для создания нового образа подразумевается, что Dockerfile находится в текущей рабочей директории. Если этот файл находится в каком-то другом месте, его расположение можно указать с использованием флага -f.  
  
Контейнеры, как мы выяснили в первом материале этой серии, состоят из слоёв. Каждый слой, кроме последнего, находящегося поверх всех остальных, предназначен только для чтения. Dockerfile сообщает системе Docker о том, какие слои и в каком порядке надо добавить в образ.  
  
Каждый слой, на самом деле, это всего лишь файл, который описывает изменение состояния образа в сравнении с тем состоянием, в котором он пребывал после добавления предыдущего слоя. В Unix, кстати, практически всё что угодно — это [файл](https://en.wikipedia.org/wiki/Everything_is_a_file).  
  
Базовый образ — это то, что является исходным слоем (или слоями) создаваемого образа. Базовый образ ещё называют родительским образом.

*Базовый образ — это то, с чего начинается образ Docker*  
  
Когда образ загружается из удалённого репозитория на локальный компьютер, то физически скачиваются лишь слои, которых на этом компьютере нет. Docker стремится экономить пространство и время путём повторного использования существующих слоёв.

### **Файлы Dockerfile**

В файлах Dockerfile содержатся инструкции по созданию образа. С них, набранных заглавными буквами, начинаются строки этого файла. После инструкций идут их аргументы. Инструкции, при сборке образа, обрабатываются сверху вниз. Вот как это выглядит:

FROM ubuntu:18.04

COPY . /app

Слои в итоговом образе создают только инструкции FROM, RUN, COPY, и ADD. Другие инструкции что-то настраивают, описывают метаданные, или сообщают Docker о том, что во время выполнения контейнера нужно что-то сделать, например — открыть какой-то порт или выполнить какую-то команду.  
  
Здесь мы исходим из предположения, в соответствии с которым используется образ Docker, основанный на Unix-подобной ОС. Конечно, тут можно воспользоваться и образом, основанным на Windows, но использование Windows — это менее распространённая практика, работать с такими образами сложнее. В результате, если у вас есть такая возможность, пользуйтесь Unix.  
  
Для начала приведём список инструкций Dockerfile с краткими комментариями.

### Дюжина инструкций Dockerfile

1. FROM — задаёт базовый (родительский) образ.
2. LABEL — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ.
3. ENV — устанавливает постоянные переменные среды.
4. RUN — выполняет команду и создаёт слой образа. Используется для установки в контейнер пакетов.
5. COPY — копирует в контейнер файлы и папки.
6. ADD — копирует файлы и папки в контейнер, может распаковывать локальные .tar-файлы.
7. CMD — описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. Аргументы могут быть переопределены при запуске контейнера. В файле может присутствовать лишь одна инструкция CMD.
8. WORKDIR — задаёт рабочую директорию для следующей инструкции.
9. ARG — задаёт переменные для передачи Docker во время сборки образа.
10. ENTRYPOINT — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются.
11. EXPOSE — указывает на необходимость открыть порт.
12. VOLUME — создаёт точку монтирования для работы с постоянным хранилищем.

*[](#_top)смотри какая крутая лягушка улыбака*

# [Doker. Цель технологии. Основные сведения. Команды. Работа с данными. Kubernetes. Сравнение Kubernetes и Docker Swarm.](#_top)

Docker – это программная платформа для быстрой разработки, тестирования и развертывания приложений. Docker упаковывает ПО в стандартизованные блоки, которые называются [контейнерами](https://aws.amazon.com/ru/containers/). Каждый контейнер включает все необходимое для работы приложения: библиотеки, системные инструменты, код и среду исполнения. Благодаря Docker можно быстро развертывать и масштабировать приложения в любой среде и сохранять уверенность в том, что код будет работать.

Использование Docker на AWS предоставляет разработчикам и системным администраторам надежный и экономичный способ сборки, доставки и запуска распределенных приложений любого масштаба.

**Почему следует использовать Docker**

Использование Docker позволяет быстрее и эффективнее доставлять или перемещать код, стандартизирует выполняемые приложениями операции и в целом экономит средства, оптимизируя использование ресурсов. Благодаря Docker пользователи получают объект, который с высокой надежностью можно запускать на любой платформе. Простой и понятный синтаксис Docker обеспечивает полный контроль над выполняемыми операциями. Повсеместное внедрение контейнеров подразумевает доступ к разнообразным инструментам и готовым приложениям, которые можно использовать с Docker.

**БЫСТРАЯ ДОСТАВКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

Пользователи Docker осуществляют доставку приложений в среднем в семь раз чаще, чем те, кто не использует Docker. Docker позволяет доставлять изолированные сервисы с необходимой периодичностью.

**СТАНДАРТИЗАЦИЯ ОПЕРАЦИЙ**

Доставка приложений в небольших контейнерах упрощает процесс развертывания, обнаружения проблем и отката для их устранения к прошлым версиям.

**ЭФФЕКТИВНОЕ ПЕРЕМЕЩЕНИЕ**

Приложения на основе контейнеров Docker можно эффективно переносить с локальных машин, на которых ведется разработка, в AWS для рабочего развертывания.

**ЭКОНОМИЯ**

Контейнеры Docker позволяют исполнять больше кода на каждом сервере, повышая эффективность использования ресурсов и сокращая расходы.

**Когда рекомендуется использовать Docker**

Контейнеры Docker можно использовать в качестве основных компонентов для создания современных платформ и приложений. Docker упрощает сборку и запуск распределенных микросервисных архитектур, развертывание кода с помощью стандартизированных конвейеров непрерывной интеграции и доставки, создание высокомасштабируемых систем обработки данных и полностью управляемых платформ для разработчиков. Сотрудничество AWS и Docker позволит упростить операцию развертывания артефактов Docker Compose в Amazon ECS и AWS Fargate.

**МИКРОСЕРВИСЫ**

Воспользуйтесь преимуществами стандартизированного развертывания кода с помощью контейнеров Docker для создания и масштабирования распределенных архитектур приложений.

**НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ И ДОСТАВКА**

Ускорьте доставку приложений за счет стандартизации рабочих сред и устранения расхождений между стеками и версиями языков программирования.

**ОБРАБОТКА ДАННЫХ**

Обработки больших данных как сервис. Объедините пакеты данных и аналитики в портативные контейнеры, запуск которых не требует специальных технических навыков.

**КОНТЕЙНЕР КАК СЕРВИС**

Создавайте и доставляйте распределенные приложения, содержание и инфраструктура которых программно управляема и защищена.

## Список полезных )) команд

Теперь приведем список полезных команд, которые могут пригодиться при работе с докером.

### Посмотреть список всех контейнеров

Эта команда выведет список всех докер контейнеров:

docker ps

Но по умолчанию выводятся только работающие контейнеры. Чтобы вывести все, в том числе и остановленные, используйте опцию -a:

docker ps -a

### Остановить и удалить все докер контейнеры

Чтобы удалить контейнеры, сначала их нужно остановить. Первая команда остановит запущенные контейнеры, если они есть. А вторая команда — удалит их.

docker stop $(docker ps -a -q)

docker rm $(docker ps -a -q)

### Запустить контейнер с последующим удалением

По умолчанию контейнеры после завершения работы останавливаются, но не удаляются. Они сохраняют свое состояние и при необходимости их можно запустить снова. Чтобы контейнер удалялся сразу после остановки, добавьте к команде docker run параметр —rm, например:

docker run --rm ubuntu echo 'hello from ubuntu'

### Посмотреть список всех скачанных образов

docker images

### Удалить докер образ

docker rmi <имя-образа>

Если у этого образа есть контейнеры, пусть даже остановленные, докер не позволит его удалить. Он выдаст сообщение:

unable to remove repository reference <имя-образа> (must force) - container <id-контейнера> is using its

Чтобы принудительно удалить образ, добавьте флаг -f:

docker rmi -f <имя-образа>

### Получить список всех контейнеров, созданных из определенного образа

docker ps -a --filter ancestor=<название-образа>

## КуБерНетЕс

*(на этом моменте оба автора вероятно сошли с ума, анализ 45 страниц полного кала хуже любого произведения лавкрафта)*

Kubernetes — это портативная расширяемая платформа с открытым исходным кодом для управления контейнеризованными рабочими нагрузками и сервисами, которая облегчает как декларативную настройку, так и автоматизацию. У платформы есть большая, быстро растущая экосистема. Сервисы, поддержка и инструменты Kubernetes широко доступны.

### Зачем вам Kubernetes и что он может сделать?

*а этот ваш куберебес нахуй нам не нужон. Это игрушка дьявола. Вы читаете его тексты. Зачем вы это делаете? Зачем вы его прославляете? Какие вы после этого последователи божьи?*

Контейнеры — отличный способ связать и запустить ваши приложения. В производственной среде необходимо управлять контейнерами, которые запускают приложения, и гарантировать отсутствие простоев. Например, если контейнер выходит из строя, необходимо запустить другой контейнер. Не было бы проще, если бы такое поведение обрабатывалось системой?

Вот тут Kubernetes приходит на помощь! Kubernetes дает вам фреймворк для гибкой работы распределенных систем. Он занимается масштабированием и обработкой ошибок в приложении, предоставляет шаблоны развертывания и многое другое. Например, Kubernetes может легко управлять канареечным развертыванием вашей системы.

Kubernetes предоставляет вам:

* **Мониторинг сервисов и распределение нагрузки** Kubernetes может обнаружить контейнер, используя имя DNS или собственный IP-адрес. Если трафик в контейнере высокий, Kubernetes может сбалансировать нагрузку и распределить сетевой трафик, чтобы развертывание было стабильным.
* **Оркестрация хранилища** Kubernetes позволяет вам автоматически смонтировать систему хранения по вашему выбору, такую как локальное хранилище, провайдеры общедоступного облака и многое другое.
* **Автоматическое развертывание и откаты** Используя Kubernetes можно описать желаемое состояние развернутых контейнеров и изменить фактическое состояние на желаемое. Например, вы можете автоматизировать Kubernetes на создание новых контейнеров для развертывания, удаления существующих контейнеров и распределения всех их ресурсов в новый контейнер.
* **Автоматическое распределение нагрузки** Вы предоставляете Kubernetes кластер узлов, который он может использовать для запуска контейнерных задач. Вы указываете Kubernetes, сколько ЦП и памяти (ОЗУ) требуется каждому контейнеру. Kubernetes может разместить контейнеры на ваших узлах так, чтобы наиболее эффективно использовать ресурсы.
* **Самоконтроль** Kubernetes перезапускает отказавшие контейнеры, заменяет и завершает работу контейнеров, которые не проходят определенную пользователем проверку работоспособности, и не показывает их клиентам, пока они не будут готовы к обслуживанию.
* **Управление конфиденциальной информацией и конфигурацией** Kubernetes может хранить и управлять конфиденциальной информацией, такой как пароли, OAuth-токены и ключи SSH. Вы можете развертывать и обновлять конфиденциальную информацию и конфигурацию приложения без изменений образов контейнеров и не раскрывая конфиденциальную информацию в конфигурации стека.

### Чем Kubernetes не является

Kubernetes ― это не традиционная комплексная система PaaS (платформа как услуга). Поскольку Kubernetes работает на уровне контейнеров, а не на уровне оборудования, у него имеется определенные общеприменимые возможности, характерные для PaaS, такие как развертывание, масштабирование, балансировка нагрузки, ведение журналов и мониторинг. Тем не менее, Kubernetes это не монолитное решение, поэтому указанные возможности по умолчанию являются дополнительными и подключаемыми. У Kubernetes есть компоненты для создания платформы разработчика, но он сохраняет право выбора за пользователем и гибкость там, где это важно.

Kubernetes:

* **Не ограничивает типы поддерживаемых приложений**. Kubernetes стремится поддерживать широкий спектр рабочих нагрузок, включая те, у которых есть или отсутствует состояние, а также связанные с обработкой данных. Если приложение может работать в контейнере, оно должно отлично работать и в Kubernetes.
* **Не развертывает исходный код и не собирает приложение**. Рабочие процессы непрерывной интеграции, доставки и развертывания (CI/CD) определяются культурой и предпочтениями организации, а также техническими требованиями.
* **Не предоставляет сервисы для приложения, такие как промежуточное программное обеспечение** (например, очереди сообщений), **платформы обработки данных** (например, Spark), **базы данных** (например, MySQL), **кеши или кластерные системы хранения** (например, Ceph), **как встроенные сервисы**. Такие компоненты могут работать в Kubernetes и/или могут быть доступны для приложений, работающих в Kubernetes, через переносные механизмы, такие как Open Service Broker.
* **Не включает решения для ведения журнала, мониторинга или оповещения**. Он обеспечивает некоторые интеграции в качестве доказательства концепции и механизмы для сбора и экспорта метрик.
* **Не указывает и не требует настройки языка/системы** (например, Jsonnet). Он предоставляет декларативный API, который может быть нацелен на произвольные формы декларативных спецификаций.
* **Не предоставляет и не принимает никаких комплексных систем конфигурации, технического обслуживания, управления или самовосстановления.**
* Кроме того, Kubernetes — **это не просто система оркестрации. Фактически, Kubernetes устраняет необходимость в этом.** Техническое определение оркестрации — это выполнение определенного рабочего процесса: сначала сделай A, затем B, затем C. Напротив, Kubernetes содержит набор независимых, компонуемых процессов управления, которые непрерывно переводит текущее состояние к предполагаемому состоянию. Неважно, как добраться от А до С. Не требуется также централизованный контроль. Это делает систему более простой в использовании, более мощной, надежной, устойчивой и расширяемой.

*В ЧЕМ ЖЕ ОТЛИИЧЕ???? ХМММ*

### Docker vs Kubernetes: чем они отличаются

Разница между Docker и Kubernetes в том, что два функционально разных инструмента для работы с контейнерами. Первый используют для создания изолированной инфраструктуры, второй — для настройки взаимодействия между несколькими контейнерами. Правильнее было бы сравнивать инструмент Docker Swarm (от разработчиков Docker) и Kubernetes. Они оба являются оркестраторами и составляют друг другу конкуренцию.

### Docker Swarm vs Kubernetes

Docker Swarm (DS) — платформа оркестровки контейнеров, работающая в среде Docker и использующая стандартный API. Kubernetes — стороннее приложение. Среда работы не единственное отличие DS и Kube, среди прочих выделяются следующие:

1. **Развертывание приложений.** Docker Swarm разворачивает приложения как сервисы в Swarm-кластере. За обозначение контейнеров отвечает файл YAML. Kubernetes выполняет развертывание приложений с помощью нескольких модулей или микросервисов.
2. **Установка платформы.** Kubernetes устанавливается в ручном режиме, без знаний и навыков здесь не обойтись. Последовательность действий может меняться в зависимости от операционной системы. Установка Docker Swarm выполняется намного легче. Для создания сред потребуется освоить набор инструментов.
3. **Простота освоения.** Пользователи с опытом работы в Docker легко разберутся в управлении DS. Kube потребует изучения.
4. **Журналирование.** Kubernetes предлагает несколько способов мониторинга и ведения журнала. В стандартную функциональность Swarm входит только мониторинг. Для больших возможностей потребуются дополнительные приложения.
5. **Масштабирование.** Kubernetes является идеальным инструментом для распределенных систем. DS быстрее разворачивает контейнеры.
6. **Сеть.** Контейнеры в Kube могут общаться друг с другом/ Docker Swarm позволяет шифровать пользовательский трафик.

Цитата «[Kubernetes](https://kubernetes.io/) — это технология, которая позволяет автоматизировать развёртывание и масштабирование контейнеризированных приложений, а также управление ими. Это — бесспорный лидер рынка средств для оркестрации контейнеров. Если вам нужен инструмент для работы с группами контейнеров, для масштабирования решений, основанных на них, используйте не Docker Swarm, а Kubernetes. Kubernetes не является частью Docker. Они с Docker, скорее, похожи на лучших друзей.»

[Ссылка на начало](#_top)